GLGJing's Blog

If you want to go fast, go alone. If you want to go far, bring others along.

C++虚函数浅析

Jan 3rd, 2015 1:59 am | Comments

一 引言

C++面向对象语言的一大特性就是抽象，在程序设计上的体现就是鼓励面向接口编程，而不要面向具体实现编程。这里所说的抽象和接口与C++的多态性密切相关。C++的多态分为静态多态（编译时多态）和动态多态（运行时多态）两大类。静态多态通过重载、模板来实现；动态多态就是通过本文的主角虚函数来体现的。虚函数是C++语言一个非常重要的特性，不同编译器对此特性的实现机制也略有差别，虽然具体实现细节由编译器说的算，在大多情况下我们不需要关心，但虚函数在某些情况下对程序的占用内存大小和执行效率有比较明显的影响，这时候知道虚函数背后的实现原理，知其然、知其所以然是很有必要的。转载请注明出处：http://glgjing.github.io/。

二 虚函数实现原理

虚函数的作用说白了就是：当调用一个虚函数时，被执行的代码必须和调用函数的对象的动态类型相一致。编译器需要做的就是如何高效的实现提供这种特性。不同编译器实现细节也不相同。大多数编译器通过vtbl（virtual table）和vptr（virtual table pointer）来实现的。当一个类声明了虚函数或者继承了虚函数，这个类就会有自己的vtbl。vtbl实际上就是一个函数指针数组，有的编译器用的是链表，不过方法都是差不多。vtbl数组中的每一个元素对应一个函数指针指向该类的一个虚函数，同时该类的每一个对象都会包含一个vptr，vptr指向该vtbl的地址。例如一个类的定义如下：

```
1 class A {
2 public:
3   virtual void Func1();
4   virtual void Func2();
5   virtual void Func3();
6   void Func4();
7 };
```

A类的对象的结构如下：

a	{...}
__vfptr	0x00a9cc78 {ConsoleApplication1.exe!const A::`vftable'} {0x00a913ca {ConsoleApplication1.exe!A::Func1(void)}, ...}
[0]	0x00a913ca {ConsoleApplication1.exe!A::Func1(void)}
[1]	0x00a9123a {ConsoleApplication1.exe!A::Func2(void)}
[2]	0x00a91136 {ConsoleApplication1.exe!A::Func3(void)}

从图中可以看出A类对象中包含一个vptr，而vptr的值就是A类的vtbl的地址，vtbl中三个元素的值分别是虚函数Func1、Func2、Func3的地址，而非虚函数Func4并没有在vtbl中。

1 单继承情况

下面定义一个类B继承自类A，重写了(override)虚函数Func1，并且定义了自己的虚函数。

```
1 class B : public A {
2 public:
3   virtual void Func1();
4   virtual void Func5();
5 };
```

声明一个A类指针A* a = new B;。a的内容如下：

__vfptr	0x0128dc8c {ConsoleApplication1.exe!const B::`vftable'} {0x01281203 {ConsoleApplication1.exe!B::Func1(void)}, ...}
[0]	0x01281203 {ConsoleApplication1.exe!B::Func1(void)}
[1]	0x0128128a {ConsoleApplication1.exe!A::Func2(void)}
[2]	0x01281154 {ConsoleApplication1.exe!A::Func3(void)}
[3]	0x0128113b {ConsoleApplication1.exe!B::Func5(void)}

同样a的值包含一个vptr，vptr指向B类的vtbl，而B类的vtbl中元素的值相比A类的vtbl有一些变化，B类重写A类虚函数的地址B::Func1取代了A::Func1在vtbl的位置，B类新定义的虚函数B::Func5也被添加到vtbl中，且父类的虚函数在子类的虚函数前面。此时通过指针a调用Func1过程相当于：

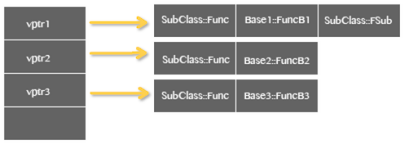
1. 通过指针a指向的地址，取出的具体值为类型B，找到里面的vptr，该vptr指向的是B类的vtbl
2. 通过vptr找到B类的vtbl地址
3. 然后通过B类的vtbl找到Func1的函数地址，调用执行，最终执行的就是B::Func1

以上过程就基本解释了C++虚函数是如何做到被执行的代码和调用函数的对象的动态类型相一致的特性了。

2 多继承情况

上面讨论的是单继承的情况，下面来看一下多继承情况。定义三个类Base1Base2Base3，和一个子类SubClass如下：

```
1  class Base1 {
2  public:
3    virtual void Func() {}
4    virtual void FuncB1() {}
5  };
6
7  class Base2 {
8  public:
9    virtual void Func() {}
10   virtual void FuncB2() {}
11 };
12
13 class Base3 {
14 public:
15   virtual void Func() {}
16   virtual void FuncB3() {}
17 };
18
19 class SubClass : public Base1, public Base2, public Base3 {
20 public:
21   virtual void Func() {}
22   virtual void FuncSub() {}
23 };
```



`SubClass`对象的结构如下：可以看到`SubClass`对象包含三个vptr分别指向三个vtbl，每个vtbl对应一个父类的vtbl。当子类覆盖父类的虚函数时，对应的vtbl变化与单继承情况相同，就是子类的虚函数替换父类相应虚函数在vtbl的位置。子类新定义的虚函数，会放在第一个父类的vtbl的后面，这里的第一个是指继承类的顺序（其实这个并不是一定的，不同编译器有自由去选择不同的实现方式）。之所以这样设计就是为了解决不同的父类类型指针在指向同一个子类实例，能够调用到各自实际的虚函数。具体的调用过程同单继承的情况相同。

结论：

- 每个声明了虚函数或者继承了虚函数的类，都会有一个自己的vtbl
- 同时该类的每个对象都会包含一个vptr去指向该vtbl
- 虚函数按照其声明顺序放于vtbl表中，vtbl数组中的每一个元素对应一个函数指针指向该类的虚函数
- 如果子类覆盖了父类的虚函数，将被放到了虚表中原来父类虚函数的位置
- 在多继承的情况下，每个父类都有自己的虚表。子类的成员函数被放到了第一个父类的表中

三 虚函数所需的代价

上面介绍了虚函数的基本实现原理，虚函数的优点不用多说，实现了运行时多态的特性。下面来分析下虚函数所需的代价，程序运行时代价无非主要体现在时间和空间上。

调用性能方面

从前面虚函数的调用过程可知。当调用虚函数时过程如下（引自More Effective C++）：

1. 通过对象的 vptr 找到类的 vtbl。这是一个简单的操作,因为编译器知道在对象内 哪里能找到 vptr(毕竟是由编译器放置的它们)。因此这个代价只是一个偏移调整(以得到 vptr)和一个指针的间接寻址(以得到 vtbl)。
2. 找到对应 vtbl 内的指向被调用函数的指针。这也是很简单的, 因为编译器为每个虚函数在 vtbl 内分配了一个唯一的索引。这步的代价只是在 vtbl 数组内 的一个偏移。
3. 调用第二步找到的的指针所指向的函数。

在单继承的情况下，调用虚函数所需的代价基本上和非虚函数效率一样，在大多数计算机上它多执行了很少的一些指令，所以有很多人一概而论说虚函数性能不行是不太科学的。在多继承的情况下，由于会根据多个父类生成多个vptr，在对象里为寻找 vptr 而进行的偏移量计算会变得复杂一些，但这些并不是虚函数的性能瓶颈。虚函数运行时所需的代价主要是虚函数不能是内联函。这也是非常好理解的，是因为内联函数是指在编译期间用被调用的函数体本身来代替函数调用的指令，但是虚函数的"虚"是指"直到运行时才能知道要调用的是哪一个函数。"但虚函数的运行时多态特性就是要在运行时才知道具体调用哪个虚函数，所以没法在编译时进行内联函数展开。当然如果通过对象直接调用虚函数它是可以被内联，但是大多数虚函数是通过对象的指针或引用被调用的，这种调用不能被内联。因为这种调用是标准的调用方式，所以虚函数实际上不能被内联。

占用空间方面

在上面的虚函数实现原理部分，可以看到为了实现运行时多态机制，编译器会给每一个包含虚函数或继承了虚函数的类自动建立一个虚函数表，所以虚函数的一个代价就是会增加类的体积。在虚函数接口较少的类中这个代价并不明显，虚函数表vtbl的体积相当于几个函数指针的体积，如果你有大量的类或者在每个类中有大量的虚函数,你会发现 vtbl 会占用大量的地址空间。但这并不是最主要的代价，主要的代价是发生在类的继承过程中，在上面的分析中，可以看到，当子类继承父类的虚函数时，子类会有自己的vtbl，如果子类只覆盖父类的一两个虚函数接口，子类vtbl的其余部分内容会与父类重复。这在如果存在大量的子类继承，且重写父类的虚函数接口只占总数的一小部分的情况下，会造成大量地址空间浪费。在一些GUI库上这种大量子类继承自同一父类且只覆盖其中一两个虚函数的情况是经常有的，这样就导致UI库的占用内存明显变大。由于虚函数指针vptr的存在，虚函数也会增加该类的每个对象的体积。在单继承或没有继承的情况下，类的每个对象会多一个vptr指针的体积，也就是4个字节；在多继承的情况下，类的每个对象会多N个（N＝包含虚函数的父类个数）vptr的体积，也就是4N个字节。当一个类的对象体积较大时，这个代价不是很明显，但当一个类的对象很轻量的时候，如成员变量只有4个字节，那么再加上4（或4N）个字节的vptr，对象的体积相当于翻了1（或N）倍，这个代价是非常大的。

四 总结

本文主要介绍了虚函数的实现机制，以及实现该机制所付出的代价，这里没有进一步讨论继承的利与弊，但经过上面的一些讨论，也可以从侧面反映出一些问题。理解虚函数的代价是有必要的，一方面是有利于高效恰当的使用它，一方面也该意识到如果你需要这些功能,不管采取什么样的方法你都得为此付出代价，在多数情况下,你的人工模拟可能比编译器生成的代码效率更低,稳定性更差。例如使用嵌套的`switch`语句或层叠的`if-then-else`语句模拟虚函数的调用,其产生的代码比虚函数的调用还要多,而且代码运行速度也更慢。再有你必须自己人工跟踪对象类型,这意味着对象会携带它们自己的类型标签(type tag)，因此你不会得到更小的对象。

Authored by GLGJing Jan 3rd, 2015 1:59 am c++

Tweet

« c++重载、重写、重定义区别 git 修改提交历史 »

Comments